Tidy Data

## Contents



# 1 Review

So far we have:

- learned how to load in packages.
- learned how to create some plots with built in datasets.
- Some basics of coding.
- Basics of naming objects.
- General principals of calling functions.
- Transform data.
- Read in data from various sources.

This week we will learn:

- How to read in .csv files.
- How to read in excel and other data types.
- How to export data.

Overview of the class:

In this class we'll focus on tidyr, a package that provides a bunch of tools to help tidy up messy datasets. tidyr is one of the packages that comes with tidyverse. The main tool we'll use for tidying data is pivoting, which allows you to change the form of data, without changing any of the values. You can learn more about the history and theoretical underpinnings in the Tidy Data paper published in the Journal of Statistical Software.

```
library(tidyverse)
```

# 2 Tidy data

You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. Each dataset shows the same values of four variables: *country*, *year*, *population*, and *cases* of TB (tuberculosis), but each dataset organizes the values in a different way.

```
table1
```

```
## # A tibble: 6 x 4
```

```
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```r
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```r
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```r
# Spread across two tibbles
table4a # cases
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```r
table4b # population
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One of them, `table1`, will be much easier to work with inside the tidyverse because it's tidy.

There are three interrelated rules that make a dataset tidy:

1. Each variable is a column; each column is a variable.
2. **Each observation is row; each row is an observation.**
3. Each value is a cell; each cell is a single value.

@fig-tidy-structure shows the rules visually.

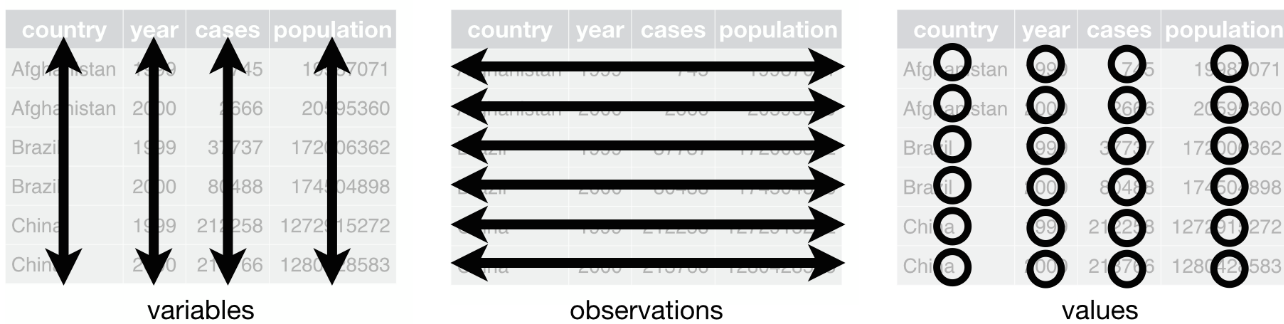



Figure 1: The following three rules make a dataset tidy: variables are columns, observations are rows, and values are cells.

Why ensure that your data is tidy? There are two main advantages:

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

2. There's a specific advantage to placing variables in columns because it allows R's vectorised nature to shine. Most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

dplyr, ggplot2, and all the other packages in the tidyverse are designed to work with tidy data. Here are a couple of small examples showing how you might work with `table1`.

```
# Compute rate per 10,000
table1 |>
  mutate(
    rate = cases / population * 10000
  )
```

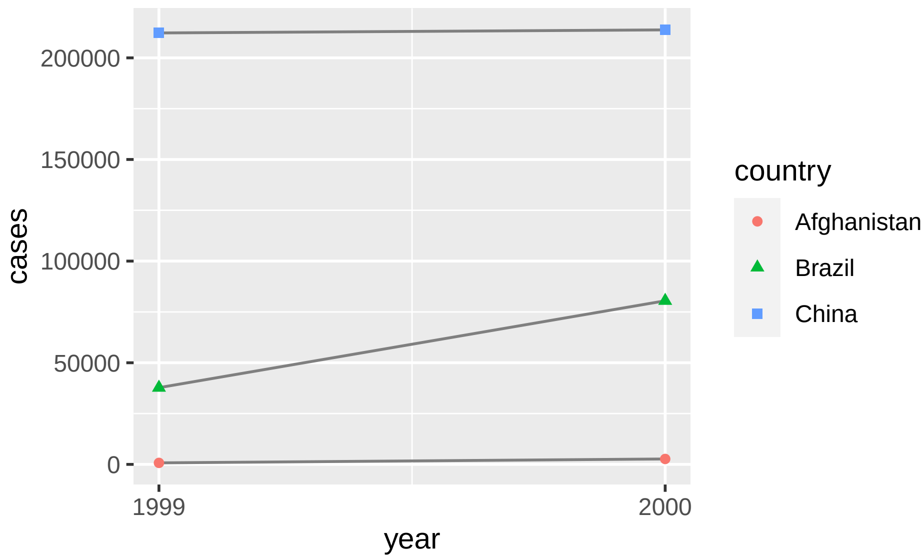
```
## # A tibble: 6 x 5
##   country      year  cases population  rate
##   <chr>       <int>  <int>      <int> <dbl>
## 1 Afghanistan  1999    745   19987071 0.373
## 2 Afghanistan  2000   2666   20595360 1.29
## 3 Brazil       1999  37737  172006362 2.19
## 4 Brazil       2000  80488  174504898 4.61
## 5 China        1999 212258 1272915272 1.67
## 6 China        2000 213766 1280428583 1.67
```

```
# Compute cases per year
table1 |>
```

```
  count(year, wt = cases)
```

```
## # A tibble: 2 x 2
##    year      n
##   <int>  <int>
## 1  1999 250740
## 2  2000 296920
```

```
# Visualise changes over time
ggplot(table1, aes(year, cases)) +
  geom_line(aes(group = country), color = "grey50") +
  geom_point(aes(color = country, shape = country)) +
  scale_x_continuous(breaks = c(1999, 2000))
```




## 3 Pivoting

The principles of tidy data might seem so obvious that you wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most real data is untidy. There are two main reasons:

1. Data is often organised to facilitate some goal other than analysis. For example, it's common for data to be structured to make data entry, not analysis, easy.
2. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a lot of time working with data.

This means that most real analyses will require at least a little tidying. You'll begin by figuring out what the underlying variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. Next, you'll **pivot** your data into a tidy form, with variables in the columns and observations in the rows.

tidyr provides two functions for pivoting data: `pivot_longer()`, which makes datasets **longer** by increasing rows and reducing columns, and `pivot_wider()` which makes datasets **wider** by increasing columns and reducing rows. The following sections work through the use of `pivot_longer()` and `pivot_wider()` to tackle a wide range of realistic datasets. These examples are drawn from `vignette("pivot", package = "tidyr")`, which you should check out if you want to see more variations and more challenging problems.

Let's dive in.

## 3.1 Data in column names

The `billboard` dataset records the billboard rank of songs in the year 2000:

```
billboard
```

```
## # A tibble: 317 x 79
##    artist     track date.entered   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8
##    <chr>      <chr> <date>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 2 Pac      Baby~ 2000-02-26      87    82    72    77    87    94    99    NA
##  2 2Ge+her    The ~ 2000-09-02      91    87    92    NA    NA    NA    NA    NA
##  3 3 Doors D~ Kryp~ 2000-04-08      81    70    68    67    66    57    54    53
##  4 3 Doors D~ Loser 2000-10-21      76    76    72    69    67    65    55    59
##  5 504 Boyz   Wobb~ 2000-04-15      57    34    25    17    17    31    36    49
##  6 98^0       Give~ 2000-08-19      51    39    34    26    26    19     2     2
##  7 A*Teens    Danc~ 2000-07-08      97    97    96    95   100    NA    NA    NA
##  8 Aaliyah    I Do~ 2000-01-29      84    62    51    41    38    35    35    38
##  9 Aaliyah    Try ~ 2000-03-18      59    53    38    28    21    18    16    14
## 10 Adams, Yo~ Open~ 2000-08-26      76    76    74    69    68    67    61    58
## # ... with 307 more rows, and 68 more variables: wk9 <dbl>, wk10 <dbl>,
## #   wk11 <dbl>, wk12 <dbl>, wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>,
## #   wk17 <dbl>, wk18 <dbl>, wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>,
## #   wk23 <dbl>, wk24 <dbl>, wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>,
## #   wk29 <dbl>, wk30 <dbl>, wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>,
## #   wk35 <dbl>, wk36 <dbl>, wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>,
## #   wk41 <dbl>, wk42 <dbl>, wk43 <dbl>, wk44 <dbl>, wk45 <dbl>, wk46 <dbl>, ...
```

In this dataset, each observation is a song. The first three columns (`artist`, `track` and `date.entered`) are variables that describe the song. Then we have 76 columns (`wk1`-`wk76`) that describe the rank of the song in each week. Here, the column names are one variable (the `week`) and the cell values are another (the `rank`).

To tidy this data, we'll use `pivot_longer()`. After the data, there are three key arguments:

- `cols` specifies which columns need to be pivoted, i.e. which columns aren't variables. This argument uses the same syntax as `select()` so here we could use `!c(artist, track, date.entered)` or `starts_with("wk")`.
- `names_to` names of the variable stored in the column names, here `"week"`.
- `values_to` names the variable stored in the cell values, here `"rank"`.

That gives the following call:

```
billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank"
  )
```

```
## # A tibble: 24,092 x 5
##    artist track                   date.entered week   rank
##    <chr>  <chr>                   <date>       <chr> <dbl>
##  1 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk1      87
##  2 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk2      82
##  3 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk3      72
##  4 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk4      77
##  5 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk5      87
##  6 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk6      94
##  7 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk7      99
```

```
##  8 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk8      NA
##  9 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk9      NA
## 10 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk10     NA
## # ... with 24,082 more rows
```

What happens if a song is in the top 100 for less than 76 weeks? Take 2 Pac's "Baby Don't Cry", for example. The above output suggests that it was only the top 100 for 7 weeks, and all the remaining weeks are filled in with missing values. These `NA`s don't really represent unknown observations; they're forced to exist by the structure of the dataset[1], so we can ask `pivot_longer()` to get rid of them by setting `values_drop_na = TRUE`:

```
billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE
  )
```

```
## # A tibble: 5,307 x 5
##    artist  track                   date.entered week   rank
##    <chr>   <chr>                   <date>       <chr> <dbl>
##  1 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk1      87
##  2 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk2      82
##  3 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk3      72
##  4 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk4      77
##  5 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk5      87
##  6 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk6      94
##  7 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk7      99
##  8 2Ge+her The Hardest Part Of ... 2000-09-02   wk1      91
##  9 2Ge+her The Hardest Part Of ... 2000-09-02   wk2      87
## 10 2Ge+her The Hardest Part Of ... 2000-09-02   wk3      92
## # ... with 5,297 more rows
```

You might also wonder what happens if a song is in the top 100 for more than 76 weeks? We can't tell from this data, but you might guess that additional columns `wk77`, `wk78`, ... would be added to the dataset.

This data is now tidy, but we could make future computation a bit easier by converting `week` into a number using `mutate()` and `parse_number()`. `parse_number()` is a function that drops any non-numeric characters before or after the first number.

```
billboard_tidy <- billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE
  ) |>
  mutate(
    week = parse_number(week)
  )
billboard_tidy
```

```
## # A tibble: 5,307 x 5
##    artist  track                   date.entered  week  rank
##    <chr>   <chr>                   <date>       <dbl> <dbl>
```

[1] We'll come back to this idea in [Chapter -@sec-missing-values].

```
##  1 2 Pac   Baby Don't Cry (Keep... 2000-02-26      1    87
##  2 2 Pac   Baby Don't Cry (Keep... 2000-02-26      2    82
##  3 2 Pac   Baby Don't Cry (Keep... 2000-02-26      3    72
##  4 2 Pac   Baby Don't Cry (Keep... 2000-02-26      4    77
##  5 2 Pac   Baby Don't Cry (Keep... 2000-02-26      5    87
##  6 2 Pac   Baby Don't Cry (Keep... 2000-02-26      6    94
##  7 2 Pac   Baby Don't Cry (Keep... 2000-02-26      7    99
##  8 2Ge+her The Hardest Part Of ... 2000-09-02      1    91
##  9 2Ge+her The Hardest Part Of ... 2000-09-02      2    87
## 10 2Ge+her The Hardest Part Of ... 2000-09-02      3    92
## # ... with 5,297 more rows
```

Now we're in a good position to look at how song ranks vary over time by drawing a plot.

```
billboard_tidy |>
  ggplot(aes(week, rank, group = track)) +
  geom_line(alpha = 1/3) +
  scale_y_reverse()
```

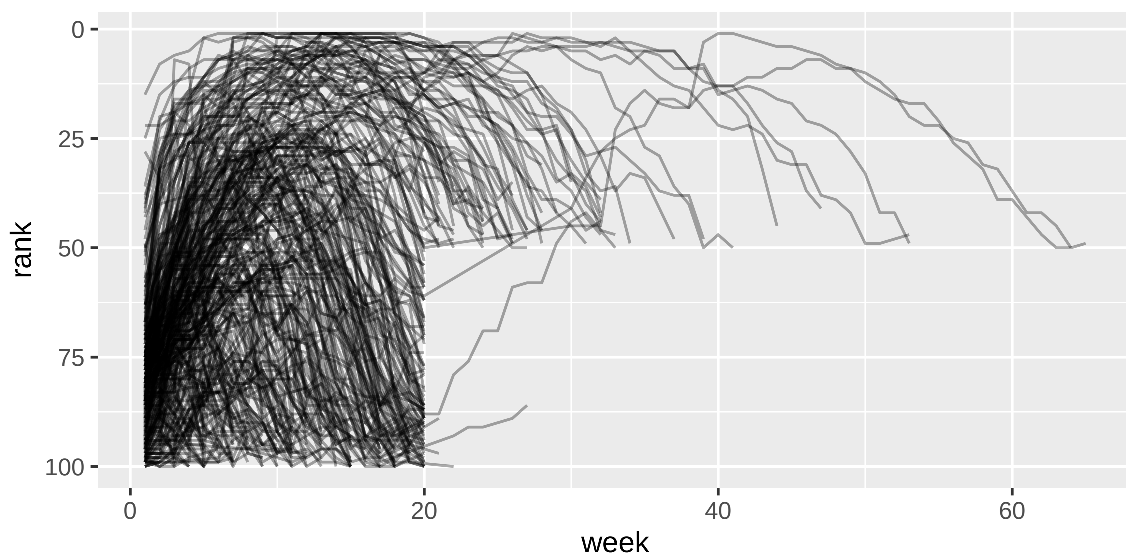



Figure 2: A line plot showing how the rank of a song changes over time.

### 3.1.1 How does pivoting work?

Now that you've seen what pivoting can do for you, it's worth taking a little time to gain some intuition about what it does to the data. Let's start with a very simple dataset to make it easier to see what's happening:

```
df <- tribble(
  ~var, ~col1, ~col2,
   "A",     1,     2,
   "B",     3,     4,
   "C",     5,     6
)
```

Here we'll say there are three variables: `var` (already in a variable), `name` (the column names in the column names), and `value` (the cell values). So we can tidy it with:

```
df |>
  pivot_longer(
    cols = col1:col2,
    names_to = "names",
    values_to = "values"
  )
```

```
## # A tibble: 6 x 3
##   var   names values
##   <chr> <chr>  <dbl>
## 1 A     col1       1
## 2 A     col2       2
## 3 B     col1       3
## 4 B     col2       4
## 5 C     col1       5
## 6 C     col2       6
```

How does this transformation take place? It's easier to see if we take it component by component. Columns that are already variables need to be repeated, once for each column in `cols`, as shown in @fig-pivot-variables.

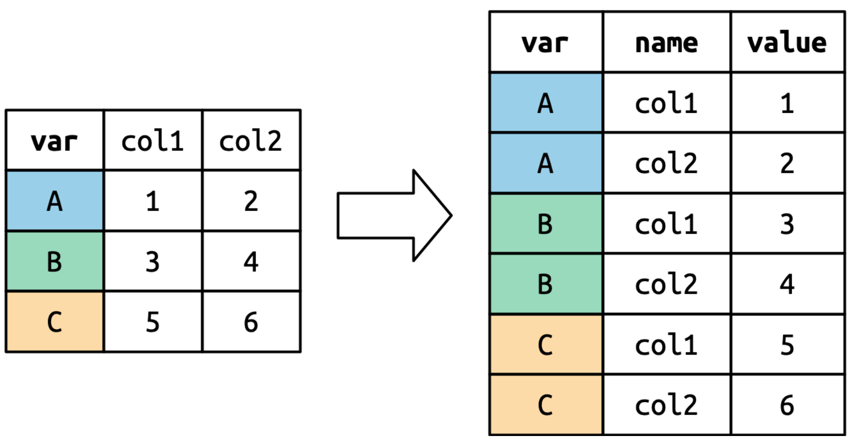



Figure 3: Columns that are already variables need to be repeated, once for each column that is pivotted.

The column names become values in a new variable, whose name is given by `names_to`.

The cell values also become values in a new variable, with a name given by `values_to`.

## 3.2 Many variables in column names

A more challenging situation occurs when you have multiple variables crammed into the column names. For example, take the `who` dataset:

```
who
```

```
## # A tibble: 7,240 x 60
##    country  iso2  iso3   year new_sp_m014 new_sp_m1524 new_sp_m2534 new_sp_m3544
##    <chr>    <chr> <chr> <int>       <int>        <int>        <int>        <int>
##  1 Afghani~ AF    AFG    1980          NA           NA           NA           NA
##  2 Afghani~ AF    AFG    1981          NA           NA           NA           NA
##  3 Afghani~ AF    AFG    1982          NA           NA           NA           NA
##  4 Afghani~ AF    AFG    1983          NA           NA           NA           NA
##  5 Afghani~ AF    AFG    1984          NA           NA           NA           NA
##  6 Afghani~ AF    AFG    1985          NA           NA           NA           NA
```

```
##   7 Afghani~ AF    AFG    1986          NA           NA           NA           NA
##   8 Afghani~ AF    AFG    1987          NA           NA           NA           NA
##   9 Afghani~ AF    AFG    1988          NA           NA           NA           NA
##  10 Afghani~ AF    AFG    1989          NA           NA           NA           NA
## # ... with 7,230 more rows, and 52 more variables: new_sp_m4554 <int>,
## #   new_sp_m5564 <int>, new_sp_m65 <int>, new_sp_f014 <int>,
## #   new_sp_f1524 <int>, new_sp_f2534 <int>, new_sp_f3544 <int>,
## #   new_sp_f4554 <int>, new_sp_f5564 <int>, new_sp_f65 <int>,
## #   new_sn_m014 <int>, new_sn_m1524 <int>, new_sn_m2534 <int>,
## #   new_sn_m3544 <int>, new_sn_m4554 <int>, new_sn_m5564 <int>,
## #   new_sn_m65 <int>, new_sn_f014 <int>, new_sn_f1524 <int>, ...
```

This data are a subset of data from the World Health Organization Global Tuberculosis Report, and accompanying global populations. who uses the original codes from the World Health Organization. The column names for columns 5 through 60 are made by combining `new_` with:

- the method of diagnosis (`rel` = relapse, `sn` = negative pulmonary smear, `sp` = positive pulmonary smear, `ep` = extrapulmonary),
- gender (`f` = female, `m` = male), and
- age group (`014` = 0-14 yrs of age, `1524` = 15-24, `2534` = 25-34, `3544` = 35-44 years of age, `4554` = 45-54, `5564` = 55-64, `65` = 65 years or older).

Each column name is made up of four pieces: three separated by `_` and the last are combined.

The first thing I'm going to do is to remove the `new_` prefix to all of the variable names. To do this I'm going to use the `names()` function. This function can be used to **get** all of the names in your data, it can also be used to change the names of your variables. First, let's see all the variable names:

```
who_a <- who
names(who_a)
```

```
##  [1] "country"      "iso2"         "iso3"         "year"         "new_sp_m014"
##  [6] "new_sp_m1524" "new_sp_m2534" "new_sp_m3544" "new_sp_m4554" "new_sp_m5564"
## [11] "new_sp_m65"   "new_sp_f014"  "new_sp_f1524" "new_sp_f2534" "new_sp_f3544"
## [16] "new_sp_f4554" "new_sp_f5564" "new_sp_f65"   "new_sn_m014"  "new_sn_m1524"
## [21] "new_sn_m2534" "new_sn_m3544" "new_sn_m4554" "new_sn_m5564" "new_sn_m65"
## [26] "new_sn_f014"  "new_sn_f1524" "new_sn_f2534" "new_sn_f3544" "new_sn_f4554"
## [31] "new_sn_f5564" "new_sn_f65"   "new_ep_m014"  "new_ep_m1524" "new_ep_m2534"
## [36] "new_ep_m3544" "new_ep_m4554" "new_ep_m5564" "new_ep_m65"   "new_ep_f014"
## [41] "new_ep_f1524" "new_ep_f2534" "new_ep_f3544" "new_ep_f4554" "new_ep_f5564"
## [46] "new_ep_f65"   "newrel_m014"  "newrel_m1524" "newrel_m2534" "newrel_m3544"
## [51] "newrel_m4554" "newrel_m5564" "newrel_m65"   "newrel_f014"  "newrel_f1524"
## [56] "newrel_f2534" "newrel_f3544" "newrel_f4554" "newrel_f5564" "newrel_f65"
```

We can see that some variables start with `new_` and others just start with `new`. To get rid of both, we're going to use the `sub()` function.

```
args(sub)
```

```
## function (pattern, replacement, x, ignore.case = FALSE, perl = FALSE,
##     fixed = FALSE, useBytes = FALSE)
## NULL
```

This function will search for a for a pattern and replace it. For example, we can get rid of all of the `new_` portions with:

```
sub("new_","",names(who_a))
```

```
##  [1] "country"      "iso2"         "iso3"         "year"         "sp_m014"
```

```
##  [6] "sp_m1524"     "sp_m2534"     "sp_m3544"     "sp_m4554"     "sp_m5564"
## [11] "sp_m65"       "sp_f014"      "sp_f1524"     "sp_f2534"     "sp_f3544"
## [16] "sp_f4554"     "sp_f5564"     "sp_f65"       "sn_m014"      "sn_m1524"
## [21] "sn_m2534"     "sn_m3544"     "sn_m4554"     "sn_m5564"     "sn_m65"
## [26] "sn_f014"      "sn_f1524"     "sn_f2534"     "sn_f3544"     "sn_f4554"
## [31] "sn_f5564"     "sn_f65"       "ep_m014"      "ep_m1524"     "ep_m2534"
## [36] "ep_m3544"     "ep_m4554"     "ep_m5564"     "ep_m65"       "ep_f014"
## [41] "ep_f1524"     "ep_f2534"     "ep_f3544"     "ep_f4554"     "ep_f5564"
## [46] "ep_f65"       "newrel_m014"  "newrel_m1524" "newrel_m2534" "newrel_m3544"
## [51] "newrel_m4554" "newrel_m5564" "newrel_m65"   "newrel_f014"  "newrel_f1524"
## [56] "newrel_f2534" "newrel_f3544" "newrel_f4554" "newrel_f5564" "newrel_f65"
```

Now we'll get the names like we would like them:

```
names(who_a) <- sub("new_","",names(who_a))
names(who_a) <- sub("new","",names(who_a))
who_a
```

```
## # A tibble: 7,240 x 60
##    country     iso2  iso3   year sp_m014 sp_m1524 sp_m2534 sp_m3544 sp_m4554
##    <chr>       <chr> <chr> <int>   <int>    <int>    <int>    <int>    <int>
##  1 Afghanistan AF    AFG    1980      NA       NA       NA       NA       NA
##  2 Afghanistan AF    AFG    1981      NA       NA       NA       NA       NA
##  3 Afghanistan AF    AFG    1982      NA       NA       NA       NA       NA
##  4 Afghanistan AF    AFG    1983      NA       NA       NA       NA       NA
##  5 Afghanistan AF    AFG    1984      NA       NA       NA       NA       NA
##  6 Afghanistan AF    AFG    1985      NA       NA       NA       NA       NA
##  7 Afghanistan AF    AFG    1986      NA       NA       NA       NA       NA
##  8 Afghanistan AF    AFG    1987      NA       NA       NA       NA       NA
##  9 Afghanistan AF    AFG    1988      NA       NA       NA       NA       NA
## 10 Afghanistan AF    AFG    1989      NA       NA       NA       NA       NA
## # ... with 7,230 more rows, and 51 more variables: sp_m5564 <int>,
## #   sp_m65 <int>, sp_f014 <int>, sp_f1524 <int>, sp_f2534 <int>,
## #   sp_f3544 <int>, sp_f4554 <int>, sp_f5564 <int>, sp_f65 <int>,
## #   sn_m014 <int>, sn_m1524 <int>, sn_m2534 <int>, sn_m3544 <int>,
## #   sn_m4554 <int>, sn_m5564 <int>, sn_m65 <int>, sn_f014 <int>,
## #   sn_f1524 <int>, sn_f2534 <int>, sn_f3544 <int>, sn_f4554 <int>,
## #   sn_f5564 <int>, sn_f65 <int>, ep_m014 <int>, ep_m1524 <int>, ...
```

Notice that we got rid of `new_` first, then we removed `new`. If we would've removed `new` first, all of the variables with `new_` would have just had `_` at the begining of their names, which would have been challenging to remove without removing the `_`'s from the other portion of the variable name.

What we want to change next, it the gender/age component. This data is going to be easier to make Tidy if we can separate these quantities with an `_`.

In this case, we can use the same tricks as above. We are lucky we can do this, which I will explain below.

```
names(who_a) <- sub("_m","_m_",names(who_a))
names(who_a) <- sub("_f","_f_",names(who_a))
who_a
```

```
## # A tibble: 7,240 x 60
##    country    iso2  iso3   year sp_m_014 sp_m_1524 sp_m_2534 sp_m_3544 sp_m_4554
##    <chr>      <chr> <chr> <int>    <int>     <int>     <int>     <int>     <int>
##  1 Afghanist~ AF    AFG    1980       NA        NA        NA        NA        NA
##  2 Afghanist~ AF    AFG    1981       NA        NA        NA        NA        NA
```

```
##   3 Afghanist~ AF    AFG    1982       NA        NA        NA        NA        NA
##   4 Afghanist~ AF    AFG    1983       NA        NA        NA        NA        NA
##   5 Afghanist~ AF    AFG    1984       NA        NA        NA        NA        NA
##   6 Afghanist~ AF    AFG    1985       NA        NA        NA        NA        NA
##   7 Afghanist~ AF    AFG    1986       NA        NA        NA        NA        NA
##   8 Afghanist~ AF    AFG    1987       NA        NA        NA        NA        NA
##   9 Afghanist~ AF    AFG    1988       NA        NA        NA        NA        NA
##  10 Afghanist~ AF    AFG    1989       NA        NA        NA        NA        NA
## # ... with 7,230 more rows, and 51 more variables: sp_m_5564 <int>,
## #   sp_m_65 <int>, sp_f_014 <int>, sp_f_1524 <int>, sp_f_2534 <int>,
## #   sp_f_3544 <int>, sp_f_4554 <int>, sp_f_5564 <int>, sp_f_65 <int>,
## #   sn_m_014 <int>, sn_m_1524 <int>, sn_m_2534 <int>, sn_m_3544 <int>,
## #   sn_m_4554 <int>, sn_m_5564 <int>, sn_m_65 <int>, sn_f_014 <int>,
## #   sn_f_1524 <int>, sn_f_2534 <int>, sn_f_3544 <int>, sn_f_4554 <int>,
## #   sn_f_5564 <int>, sn_f_65 <int>, ep_m_014 <int>, ep_m_1524 <int>, ...
```

This works well here, but when we use `sub` we need to make sure that we are only substituting the characters that we want to substitute.

I'll also mention, that another "easy" way to do the above is to use the `replace_if()` function.

```
who %>%
  rename_if(startsWith(names(.),"new_"), ~str_remove(.,"new_")) %>%
  rename_if(startsWith(names(.),"new"), ~str_remove(.,"new")) %>%
  rename_if(grepl("_m",names(.)), ~str_replace(.,"_m", "_m_")) %>%
  rename_if(grepl("_f",names(.)), ~str_replace(.,"_f", "_f_"))
```

```
## # A tibble: 7,240 x 60
##    country    iso2  iso3   year sp_m_014 sp_m_1524 sp_m_2534 sp_m_3544 sp_m_4554
##    <chr>      <chr> <chr> <int>    <int>     <int>     <int>     <int>     <int>
##   1 Afghanist~ AF    AFG    1980       NA        NA        NA        NA        NA
##   2 Afghanist~ AF    AFG    1981       NA        NA        NA        NA        NA
##   3 Afghanist~ AF    AFG    1982       NA        NA        NA        NA        NA
##   4 Afghanist~ AF    AFG    1983       NA        NA        NA        NA        NA
##   5 Afghanist~ AF    AFG    1984       NA        NA        NA        NA        NA
##   6 Afghanist~ AF    AFG    1985       NA        NA        NA        NA        NA
##   7 Afghanist~ AF    AFG    1986       NA        NA        NA        NA        NA
##   8 Afghanist~ AF    AFG    1987       NA        NA        NA        NA        NA
##   9 Afghanist~ AF    AFG    1988       NA        NA        NA        NA        NA
##  10 Afghanist~ AF    AFG    1989       NA        NA        NA        NA        NA
## # ... with 7,230 more rows, and 51 more variables: sp_m_5564 <int>,
## #   sp_m_65 <int>, sp_f_014 <int>, sp_f_1524 <int>, sp_f_2534 <int>,
## #   sp_f_3544 <int>, sp_f_4554 <int>, sp_f_5564 <int>, sp_f_65 <int>,
## #   sn_m_014 <int>, sn_m_1524 <int>, sn_m_2534 <int>, sn_m_3544 <int>,
## #   sn_m_4554 <int>, sn_m_5564 <int>, sn_m_65 <int>, sn_f_014 <int>,
## #   sn_f_1524 <int>, sn_f_2534 <int>, sn_f_3544 <int>, sn_f_4554 <int>,
## #   sn_f_5564 <int>, sn_f_65 <int>, ep_m_014 <int>, ep_m_1524 <int>, ...
```

So in this case we have five variables: two variables are already columns, three variables are contained in the column name, and one variable is in the cell name.

This requires two changes to our call to `pivot_longer()`: `names_to` gets a vector of column names and `names_sep` describes how to split the variable name up into pieces:

```
who_a |>
  pivot_longer(
    cols = !(country:year),
```

```
    names_to = c("diagnosis", "gender", "age"),
    names_sep = "_",
    values_to = "count"
  )
```

```
## # A tibble: 405,440 x 8
##    country     iso2  iso3   year diagnosis gender age   count
##    <chr>       <chr> <chr> <int> <chr>     <chr>  <chr> <int>
##  1 Afghanistan AF    AFG    1980 sp        m      014      NA
##  2 Afghanistan AF    AFG    1980 sp        m      1524     NA
##  3 Afghanistan AF    AFG    1980 sp        m      2534     NA
##  4 Afghanistan AF    AFG    1980 sp        m      3544     NA
##  5 Afghanistan AF    AFG    1980 sp        m      4554     NA
##  6 Afghanistan AF    AFG    1980 sp        m      5564     NA
##  7 Afghanistan AF    AFG    1980 sp        m      65       NA
##  8 Afghanistan AF    AFG    1980 sp        f      014      NA
##  9 Afghanistan AF    AFG    1980 sp        f      1524     NA
## 10 Afghanistan AF    AFG    1980 sp        f      2534     NA
## # ... with 405,430 more rows
```

### 3.3 Widening data

`pivot_wider()` is the opposite of `pivot_longer()`. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

To tidy this up, we first analyse the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:

- The column to take variable names from. Here, it's `type`.
- The column to take values from. Here it's `count`.

Once we've figured that out, we can use `pivot_wider()` as follows.

```
table2 |>
    pivot_wider(names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
```

```
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

## 3.4 Untidy data

While `pivot_wider()` is occasionally useful for making tidy data, its real strength is making **untidy** data. While that sounds like a bad thing, untidy isn't a pejorative term: there are many untidy data structures that are extremely useful. Tidy data is a great starting point for most analyses but it's not the only data format you'll ever need.

The following sections will show a few examples of `pivot_wider()` making usefully untidy data for presenting data to other humans, for input to multivariate statistics algorithms, and for pragmatically solving data manipulation challenges.

### 3.4.1 Presenting data to humans

As you've seen, `dplyr::count()` produces tidy data: it makes one row for each group, with one column for each grouping variable, and one column for the number of observations.

```
diamonds |>
  count(clarity, color)
```

```
## # A tibble: 56 x 3
##    clarity color     n
##    <ord>   <ord> <int>
##  1 I1      D        42
##  2 I1      E       102
##  3 I1      F       143
##  4 I1      G       150
##  5 I1      H       162
##  6 I1      I        92
##  7 I1      J        50
##  8 SI2     D      1370
##  9 SI2     E      1713
## 10 SI2     F      1609
## # ... with 46 more rows
```

This is easy to visualize or summarize further, but it's not the most compact form for display. You can use `pivot_wider()` to create a form more suitable for display to other humans:

```
diamonds |>
  count(clarity, color) |>
  pivot_wider(
    names_from = color,
    values_from = n
  )
```

```
## # A tibble: 8 x 8
##   clarity     D     E     F     G     H     I     J
##   <ord>   <int> <int> <int> <int> <int> <int> <int>
## 1 I1         42   102   143   150   162    92    50
## 2 SI2      1370  1713  1609  1548  1563   912   479
```

```
## 3 SI1      2083  2426  2131  1976  2275  1424   750
## 4 VS2      1697  2470  2201  2347  1643  1169   731
## 5 VS1       705  1281  1364  2148  1169   962   542
## 6 VVS2      553   991   975  1443   608   365   131
## 7 VVS1      252   656   734   999   585   355    74
## 8 IF         73   158   385   681   299   143    51
```

This display also makes it easy to compare in two directions, horizontally and vertically, much like `facet_grid()`.

`pivot_wider()` can be great for quickly sketching out a table. But for real presentation tables, we highly suggest learning a package like gt. gt is similar to ggplot2 in that it provides an extremely powerful grammar for laying out tables. It takes some work to learn but the payoff is the ability to make just about any table you can imagine.

### 3.4.2 Multivariate statistics

Most classical multivariate statistical methods (like dimension reduction and clustering) require your data in matrix form, where each column is a time point, or a location, or a gene, or a species, but definitely not a variable. Sometimes these formats have substantial performance or space advantages, or sometimes they're just necessary to get closer to the underlying matrix mathematics.

We're not going to cover these statistical methods here, but it is useful to know how to get your data into the form that they need. For example, let's imagine you wanted to cluster the gapminder data to find countries that had similar progression of `gdpPercap` over time. To do this, we need one row for each country and one column for each year:

```
library(gapminder)
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

```
col_year <- gapminder |>
  mutate(gdpPercap = log10(gdpPercap)) |>
  pivot_wider(
    id_cols = country,
    names_from = year,
    values_from = gdpPercap
  )
col_year
```

```
## # A tibble: 142 x 13
##    country `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992` `1997`
##    <fct>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
```

```
## 1 Afghan~    2.89   2.91   2.93   2.92   2.87   2.90   2.99   2.93   2.81   2.80
## 2 Albania    3.20   3.29   3.36   3.44   3.52   3.55   3.56   3.57   3.40   3.50
## 3 Algeria    3.39   3.48   3.41   3.51   3.62   3.69   3.76   3.75   3.70   3.68
## 4 Angola     3.55   3.58   3.63   3.74   3.74   3.48   3.44   3.39   3.42   3.36
## 5 Argent~    3.77   3.84   3.85   3.91   3.98   4.00   3.95   3.96   3.97   4.04
## 6 Austra~    4.00   4.04   4.09   4.16   4.23   4.26   4.29   4.34   4.37   4.43
## 7 Austria    3.79   3.95   4.03   4.11   4.22   4.30   4.33   4.37   4.43   4.46
## 8 Bahrain    3.99   4.07   4.11   4.17   4.26   4.29   4.28   4.27   4.28   4.31
## 9 Bangla~    2.84   2.82   2.84   2.86   2.80   2.82   2.83   2.88   2.92   2.99
## 10 Belgium   3.92   3.99   4.04   4.12   4.22   4.28   4.32   4.35   4.41   4.44
## # ... with 132 more rows, and 2 more variables: `2002` <dbl>, `2007` <dbl>
```

`pivot_wider()` produces a tibble where each row is labelled by the `country` variable. But most classic statistical algorithms don't want the identifier as an explicit variable; they want as a **row name**. We can turn the `country` variable into row names with `column_to_rowname()`:

```
col_year <- col_year |>
  column_to_rownames("country")
head(col_year)
```

```
##                 1952     1957     1962     1967     1972     1977     1982
## Afghanistan 2.891786 2.914265 2.931000 2.922309 2.869221 2.895485 2.990344
## Albania     3.204407 3.288313 3.364155 3.440940 3.520277 3.548144 3.560012
## Algeria     3.388990 3.479140 3.406679 3.511481 3.621453 3.691118 3.759302
## Angola      3.546618 3.582965 3.630354 3.742157 3.738248 3.478371 3.440429
## Argentina   3.771684 3.836125 3.853282 3.905955 3.975112 4.003419 3.954141
## Australia   4.001716 4.039400 4.086973 4.162150 4.225015 4.263262 4.289522
##                 1987     1992     1997     2002     2007
## Afghanistan 2.930641 2.812473 2.803007 2.861376 2.988818
## Albania     3.572748 3.397495 3.504206 3.663155 3.773569
## Algeria     3.754452 3.700982 3.680996 3.723295 3.794025
## Angola      3.385644 3.419600 3.357390 3.442995 3.680991
## Argentina   3.960931 3.968876 4.040099 3.944366 4.106510
## Australia   4.340224 4.369675 4.431331 4.486965 4.537005
```

This makes a data frame, because tibbles don't support row names[2].

We're now ready to cluster with (e.g.) `kmeans()`:

```
cluster <- stats::kmeans(col_year, centers = 6)
```

You can get the clustering membership out with this code:

```
cluster_id <- cluster$cluster |>
  enframe() |>
  rename(country = name, cluster_id = value)
cluster_id
```

```
## # A tibble: 142 x 2
##    country     cluster_id
##    <chr>            <int>
##  1 Afghanistan          5
##  2 Albania              1
##  3 Algeria              3
##  4 Angola               1
```

[2]tibbles don't use row names because they only work for a subset of important cases: when observations can be identified by a single character vector.

```
##  5 Argentina            6
##  6 Australia            4
##  7 Austria              4
##  8 Bahrain              4
##  9 Bangladesh           5
## 10 Belgium              4
## # ... with 132 more rows
```

You could then combine this back with the original data using one of the joins you'll learn about in later in the course:

```
gapminder |> left_join(cluster_id)
```

```
## Joining, by = "country"
```

```
## # A tibble: 1,704 x 7
##    country     continent  year lifeExp      pop gdpPercap cluster_id
##    <chr>       <fct>     <int>   <dbl>    <int>     <dbl>      <int>
##  1 Afghanistan Asia       1952    28.8  8425333      779.          5
##  2 Afghanistan Asia       1957    30.3  9240934      821.          5
##  3 Afghanistan Asia       1962    32.0 10267083      853.          5
##  4 Afghanistan Asia       1967    34.0 11537966      836.          5
##  5 Afghanistan Asia       1972    36.1 13079460      740.          5
##  6 Afghanistan Asia       1977    38.4 14880372      786.          5
##  7 Afghanistan Asia       1982    39.9 12881816      978.          5
##  8 Afghanistan Asia       1987    40.8 13867957      852.          5
##  9 Afghanistan Asia       1992    41.7 16317921      649.          5
## 10 Afghanistan Asia       1997    41.8 22227415      635.          5
## # ... with 1,694 more rows
```